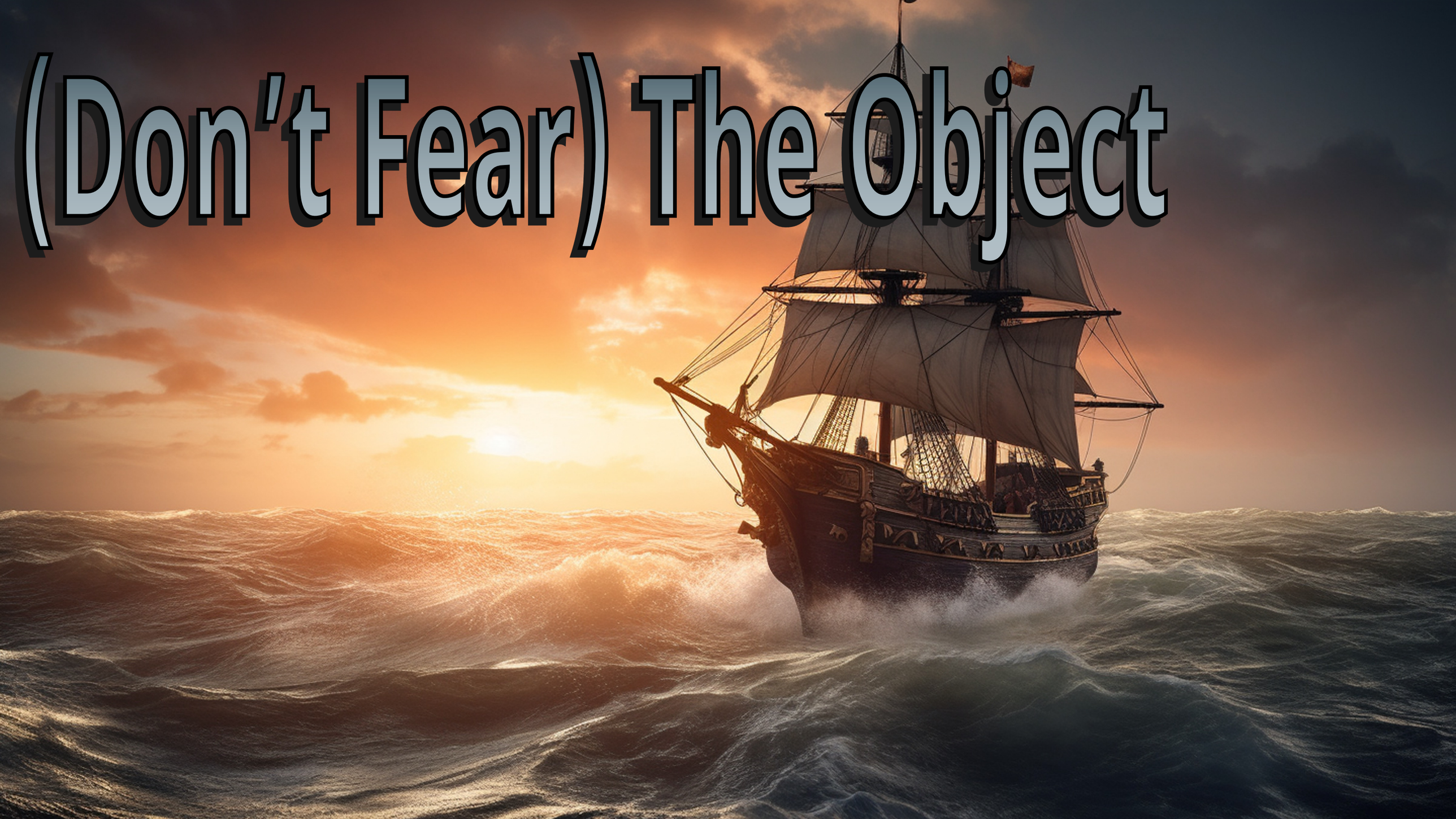
(Don't Fear) The Object



Fearless: A minimalistic nominally typed pure expression-based OO language where there are no fields and all the state is captured by closures.

The Fearless Heart: Encoding booleans, optionals, and lists.

Braving mutability: how to add mutability to such a language without losing the reasoning advantages of functional programming.

The Treasure: support for automatic parallelisation, correct cache invalidation and representation invariants.

The Fearless Heart

First, we will sail comforting friendly waters, exploring the functional core of Fearless

```
//A Person with an age and name

Person[]:{ .age[](): Num, .name[](): Str }
```

```
//A Person with an age and name

Person:{ .age: Num, .name: Str }
```

```
Person:{ .age: Num, .name: Str }  //A Person with an age and name

//making a person directly
Bob:Person{ .age: Num -> 42, .name: Str -> "Bob" }

FPerson:{ .of(age: Num, name: Str): Person -> Bob:Person{
  .age -> age,
  .name -> name
}}
FPerson.of(42, "Bob")    //making a person with a factory
```

What are 42 and "Bob"? Are they in syntax we've talked about before? Yes!

- `42` ***desugared as*** `FreshName:42[]{}`
- `"Bob"` ***desugared as*** `FreshName:"Bob"[]{}`
- `FPerson == FPerson[]{} == FreshName:FPerson[]{}`

```
Person:{ .age: Num, .name: Str }  //A Person with an age and name

//making a person directly
Bob:Person{ .age -> 42, .name -> "Bob" }
// or, a factory
FPerson:{ .of(age: Num, name: Str): Person -> Bob:Person{
  .age -> age,
  .name -> name
}}
FPerson.of(42, "Bob")    //making a person with a factory
```

What are 42 and "Bob"? Are they in syntax we've talked about before? Yes!

- `42` ***desugared as*** `FreshName:42[]{}`
- `"Bob"` ***desugared as*** `FreshName:"Bob"[]{}`
- `FPerson == FPerson[]{} == FreshName:FPerson[]{}`

```
Person:{ .age: Num, .name: Str }  //A Person with an age and name

//making a person directly
Bob:Person{ .age -> 42, .name -> "Bob" }
// or, a factory
FPerson:{ .of(age: Num, name: Str): Person -> Person{
  .age -> age,
  .name -> name
}}
FPerson.of(42, "Bob")    //making a person with a factory
```

What are 42 and "Bob"? Are they in syntax we've talked about before? Yes!

- `42` ***desugared as*** `FreshName:42[]{}`
- `"Bob"` ***desugared as*** `FreshName:"Bob"[]{}`
- `FPerson == FPerson[]{} == FreshName:FPerson[]{}`

```
Person:{ .age: Num, .name: Str }  //A Person with an age and name

//making a person directly
Bob:Person{ .age -> 42, .name -> "Bob" }
// or, a factory
FPerson:{ .of(age: Num, name: Str): Person -> {
  .age -> age,
  .name -> name
}}
FPerson.of(42, "Bob")    //making a person with a factory
```

What are 42 and "Bob"? Are they in syntax we've talked about before? Yes!

- `42` ***desugared as*** `FreshName:42[]{}`
- `"Bob"` ***desugared as*** `FreshName:"Bob"[]{}`
- `FPerson == FPerson[]{} == FreshName:FPerson[]{}`

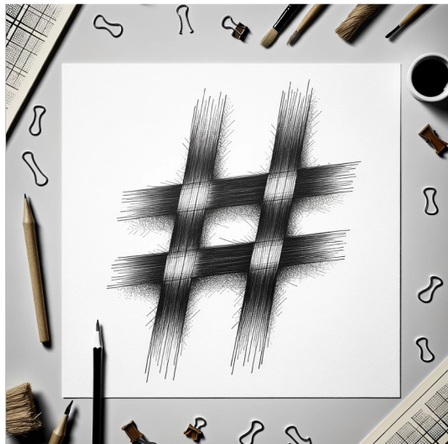
```
Person:{ .age: Num, .name: Str }  //a Person with age and name
```

Functions can just be generic top level declarations

```
F[R]:{ #: R }          //Note: # is a valid method name, just like .of
F[A, R]:{ #(a: A): R }
F[A, B, R]:{ #(a: A, b: B): R }
F[A, B, C, R]:{ #(a: A, b: B, c: C): R }
```

Now `FPerson` can be a kind of function

```
FPerson:F[Num, Str, Person]{ age, name -> {.age->age, .name->name} }

FPerson#(42, "Bob")    //making a person with a function/factory
```



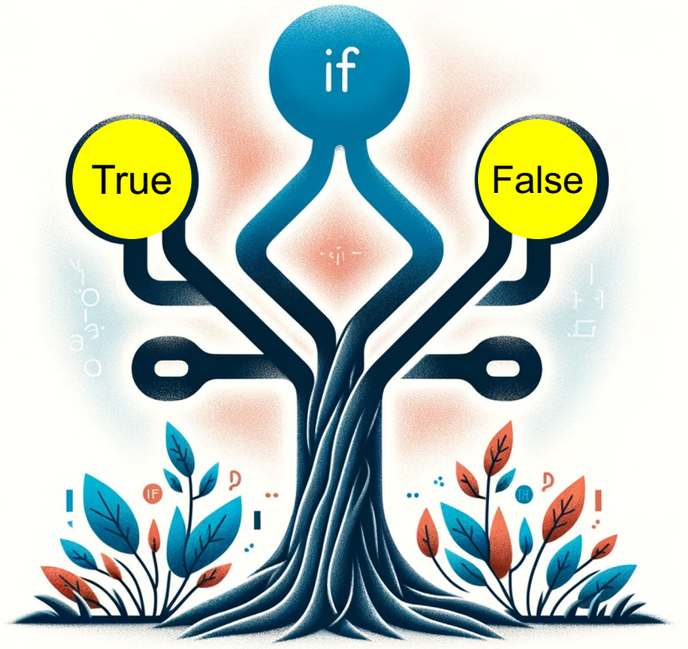
```
Person: { .age: Num, .name: Str }  //not declared at top level
```

Person as an internally declared trait instead

```
FPerson:F[Num, Str, Person]{ age, name -> Person:{
  .age:Num->age,
  .name:Str->name
  }}

FPerson#(42, "Bob")    //same instantiation syntax as before,
                       //but now this is guaranteed to
                       //be the only way to make a Person.
//A declaration name introduced inside a method body is ‘final’ and
//can not be inherited. Thus writing ‘Person{…}’ anywhere else would
//be a type error.
```

```
Bool: {
   .and(other: Bool): Bool,
   .or(other: Bool): Bool,
   .not: Bool,
   .if[R](m: ThenElse[R]): R
   }
ThenElse[R]:{ .then: R, .else: R }

True: Bool{
   .and(other) -> other,
   .or(other) -> this,
   .not -> False,
   .if(m) -> m.then,
   }
False:Bool{
   .and(other) -> this,
   .or(other) -> other,
   .not -> True,
   .if(m) -> m.else,
   }
```

```
//usage example
True.and(False).if({
   .then->/*code for the then case*/,
   .else->/*code for the else case*/,
   })

True.and False.if{
   .then->/*code for the then case*/,
   .else->/*code for the else case*/,
   }
```



```
Opt[T]: {
  .match[R](m: OptMatch[T,R]): R -> m.empty
  }
OptMatch[T,R]: {
  .empty: R,
  .some(t: T): R
  }
Opt: {
  #[T](t:T):Opt[T] -> { m -> m.some(t) }
  }
```

```
//usage example
Opt#bob       //Bob is here
Opt[Person]   //no one is here
```

```
List[T]: {
  .match[R](m: ListMatch[T,R]): R -> m.empty
  +(e: T): List[T] -> { m -> m.elem(this, e) },
}
ListMatch[T,R]: {
  .empty: R,
  .elem(list: List[T], e: T): R
}

//usage examples
List[Num]+1+2+3
List[Opt[Num]]+{}+{}+(Opt#3)
List[List[Num]]+{}+{}+(List[Num]+3)

Example: {
  .sum(ns: List[Num]): Num -> ns.match{
    .empty -> 0,
    .elem(list, e) -> this.sum(list) + e
  }
}
```

```
List[T]: {
   .match[R](m: ListMatch[T,R]): R -> m.empty
   +(e: T): List[T] -> { m -> m.elem(this, e) },
   .map[R](f: F[T, R]): List[R] -> this.match{
     .empty -> {},
     .elem(list, e) -> list.map(f) + (f#e)
    }
}
ListMatch[T,R]: {
   .empty: R,
   .elem(list: List[T], e: T): R
}
```

Braving Mutability: Fearless's Journey into mutability

Isolated iso

Mutable mut

Immutable imm

Readable read

Mutable mut

Easy to eat, easy to digest.
Hard to manage since they can tangle.

For hygienic reasons is better to eat your own food instead of having many strangers eating from the same plate.

Isolated iso

The gold standard.
Affine: can be used only zero or one times.
It can be sold to a stranger.
It is often open only in private.

The whole MROG of the iso is only reachable from the iso reference itself.

Immutable objects can be freely shared

The unchanging eternal diamond.

Deeply immutable objects as
in functional programming.

Can be shown to strangers, can be shared.

Immutable imm

The magnifying lens allows us to see stuff well, but we can not modify it or touch it using it.

Readable read


RC+OC = determinism


The Treasure:
Invariants, caching and automatic parallelism

Automatic parallelisation

```
// With parallelism
Example:{
  // Fearless could safely parallelise the two branches of this method call:
  // .fork(g1: iso Graph, g2: iso Graph): Num -> (g1.partition) + (g2.partition),
   // Currently we do it like this:
   .fork(g1: iso Graph, g2: iso Graph): Num -> this.join(
      IsoPod#g1.asyncMutate{g -> g.partition},
      IsoPod#g2.asyncMutate{g -> g.partition}),

   .join(f1: mut Future[Num], f2: mut Future[Num]): Num -> (f1#) + (f2#),
}
```

Automatic parallelisation

```
// With parallelism
Example:{
  // Fearless could safely parallelise the two branches of this method call:
  // .fork(g1: iso Graph, g2: iso Graph): Num -> (g1.partition) + (g2.partition),
  // Currently we do it like this:
  .fork(g1: iso Graph, g2: iso Graph): Num -> this.join(
     IsoPod#g1.asyncMutate{g -> g.partition},
     IsoPod#g2.asyncMutate{g -> g.partition}),

  .join(f1: mut Future[Num], f2: mut Future[Num]): Num -> (f1#) + (f2#),
}
```


Is this treasure just the tip of the iceberg?
What other properties could be enforced?
What other unobservable optimisations could be unlocked?